Automata and Computability

Prof. Michele Amoretti

High Performance Computing 2022/2023

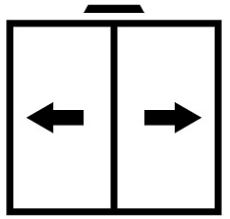
Summary

- Finite Automata
- Pushdown Automata
- Turing Machines
- Decidability
- Reducibility

Finite Automata

Finite automata are good models for computers with an extremely limited amount of memory.

Example: the controller of an automatic door



Finite automata can be either **deterministic** or **nondeterministic**.

We start with deterministic ones: when the machine is in a given state and reads the next input symbol, we know what the next state will be.

Deterministic Finite Automaton (DFA)

Formally, a DFA is a 5-tuple (Q,Σ,δ,q_0,F) where

1. Q is a finite set called the **states**
2. Σ is a finite set called the **alphabet**
3. $\delta: Q \times \Sigma \rightarrow Q$ is the **transition function**
4. $q_0 \in Q$ is the **start state**
5. $F \subseteq Q$ is the set of **accept states**

Intuitively, a DFA represents a system that can be in a finite number of different states.

As a consequence of some input (in a finite alphabet), the DFA makes a transition from the current state to another.

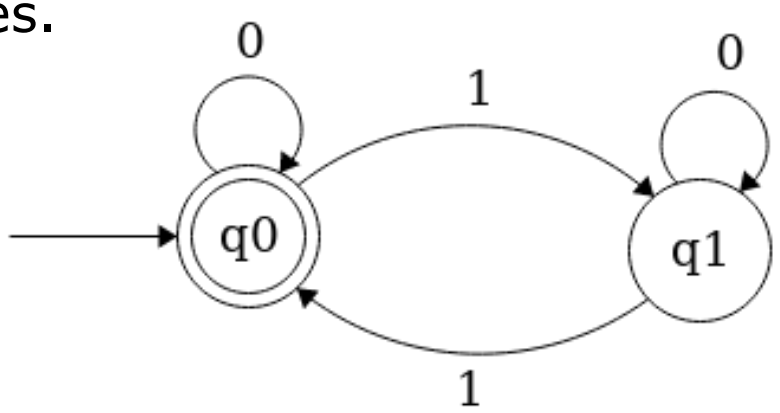
The output is *accept* if, after reading the last input, the DFA enters an accept state, and *reject* if it does not.

Deterministic Finite Automaton (DFA)

Current State	Next State (δ)	
	0	1
q_0	q_1	q_2
q_1	q_2	q_1
q_2	q_2	q_0

The DFA can be represented by means of a table that is called **state transition table**.

A much more suggestive representation of a DFA is the **state diagram**, in which the automaton is represented by means of an oriented graph, where nodes are states and arcs are transitions, the latter being labeled with the symbol whose reading causes the transition itself. The initial state is a node with an entering arrow, while accept states are double-circled nodes.



Deterministic Finite Automaton (DFA)

Let $M = (Q,\Sigma,\delta,q_0,F)$ be a DFA.

If A is the set of all strings that M accepts, we say that A is the **language of machine M** and write $L(M) = A$.

We say that **M recognizes A**.

A DFA may accept several strings, but it always recognizes only one language.

A language is called a **regular language** if some DFA recognizes it.

Finite State Transducer (FST)

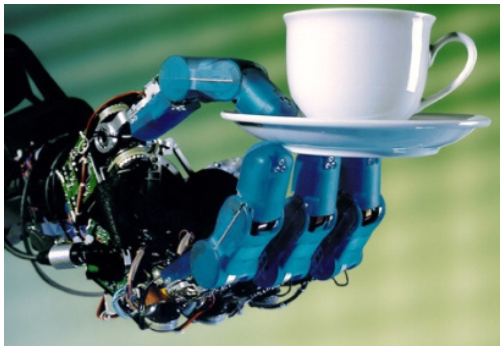
An FST is a DFA whose output is a string and not just *accept* or *reject*.

Formally, an FST is a 7-tuple $(Q,\Sigma,\delta,q_0,F,O,\eta)$ where

1. Q is a finite set called the **states**
2. Σ is a finite set called the **alphabet**
3. $\delta: Q \times \Sigma \rightarrow Q$ is the **transition function**
4. $q_0 \in Q$ is the **start state**
5. $F \subseteq Q$ is the set of **accept states**
6. O is the finite set of **output symbols**
7. $\eta: Q \times \Sigma \rightarrow O$ is the **output function**

Finite State Transducer (FST)

Example



A manufacturer robot receives saucers and small cups on an input conveyor belt, and puts assembled "small cup on saucer" on an output conveyor belt.

Rules:

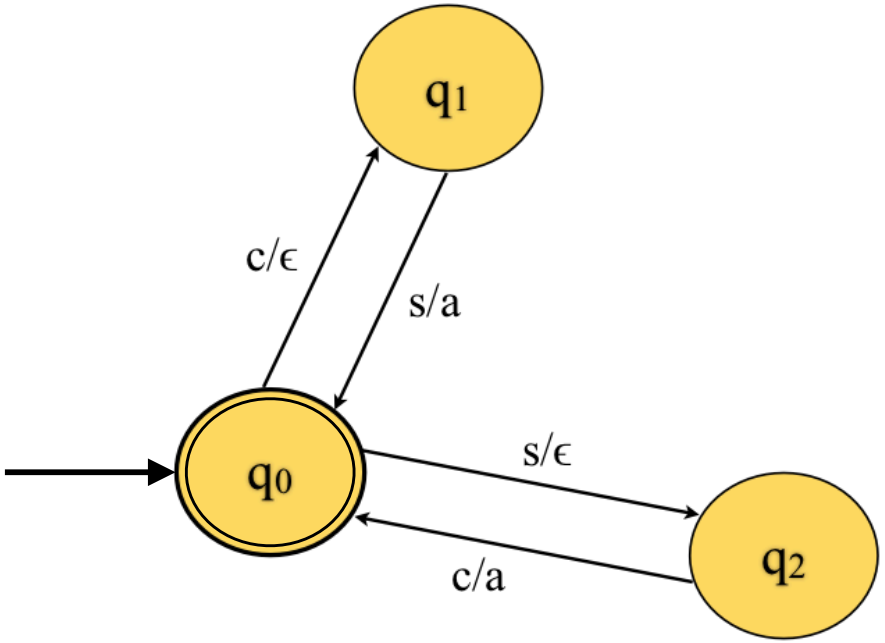
1. If both hands are free, the robot grasps the input object (either small cup or saucer) with its left hand.
2. If the robot already has an object in its left hand, the new incoming object must be different and it is grasped with the right hand.
3. The robot puts the small cup on the saucer and puts the assembled item on the output conveyor belt.
4. When the input sequence ends, both hands of the robot must be free.

Finite State Transducer (FST)

Example

A manufacturer robot receives saucers and small cups on an input conveyor belt, and puts assembled "small cup on saucer" on an output conveyor belt.

FST that models the robot:



c = small cup

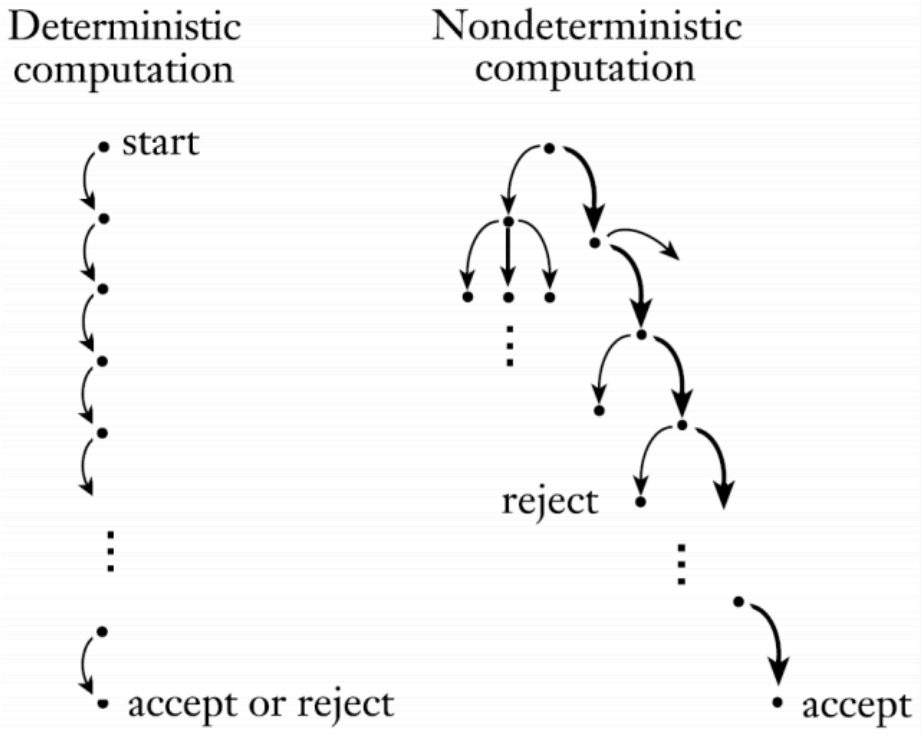
s = saucer

a = assembled item

Nondeterminism

Nondeterminism is a generalization of determinism.

In a nondeterministic machine, several choices may exist for the next state at any point.



Nondeterministic Finite Automaton (NFA)

Formally, an NFA is a 5-tuple (Q,Σ,δ,q_0,F) where

1. Q is a finite set of **states**
2. Σ is a finite **alphabet**
3. $\delta: Q \times \Sigma_\varepsilon \rightarrow P(Q)$ is the **transition function**
4. $q_0 \in Q$ is the **start state**
5. $F \subseteq Q$ is the **set of accept states**

where $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ and $P(Q)$ is the power set of Q, i.e., the collection of all subsets of Q.

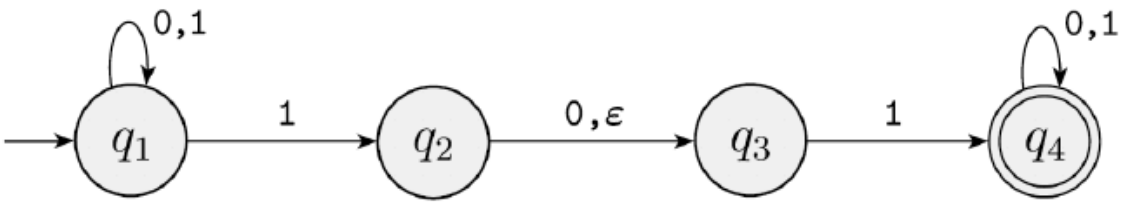
The symbol ε represents the empty string.

The NFA takes a state and an input symbol or the empty string and produces **the set of possible next states**.

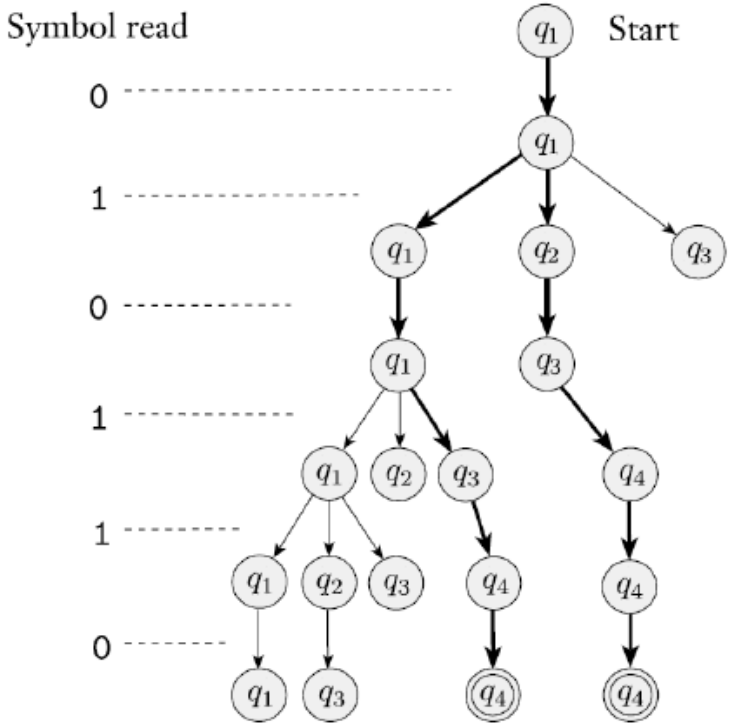
Nondeterministic Finite Automaton (NFA)

Example

N_1



Input 010110



After reading an input symbol, if from current state there are multiple ways to proceed, the machine splits into multiple copies of itself and follows all the possibilities in parallel. During the computation, some copies of the machine die.
If any one of the copies of the machine is in an accept state at the end of the input, the NFA accepts the input string.

$q_4 \in F$, thus N_1 accepts 010110

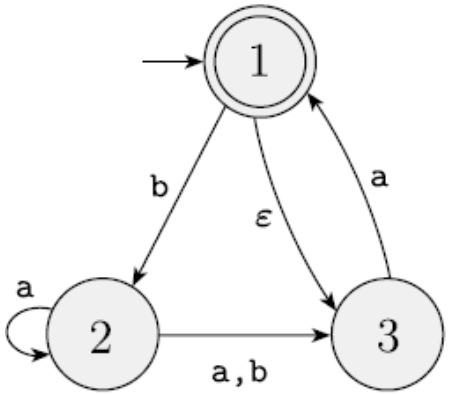
Equivalence of NFAs and DFAs

Surprisingly, DFAs and NFAs recognize the same class of languages.

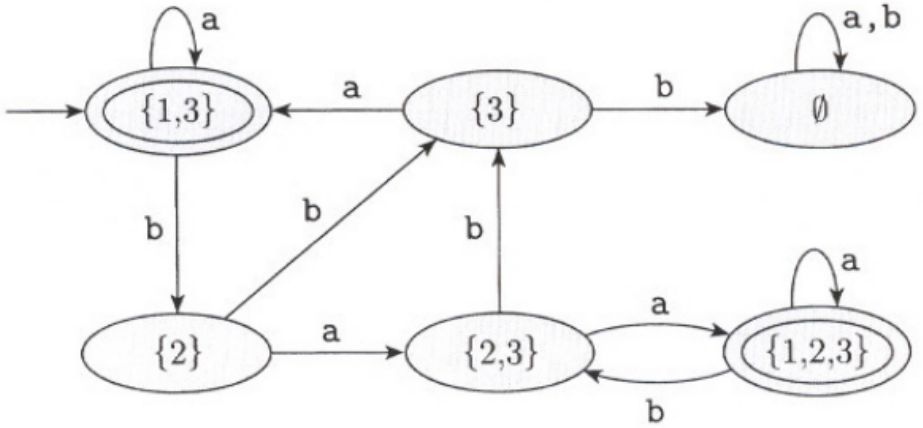
A language is called a **regular language** if some NFA recognizes it.

It is useful because describing an NFA for a given language sometimes is much easier than describing a DFA for that language.

Example



NFA



DFA

Pushdown Automata

Many languages cannot be described by means of finite automata.

Pushdown automata are like finite automata but have an extra component called a **stack**.

The stack provides extra additional memory beyond the finite amount available in the control, allowing pushdown automata to recognize some nonregular languages.

Like finite automata, pushdown automata can be **deterministic** or **nondeterministic**.

Deterministic Pushdown Automaton (DPA)

A DPA is an DFA enriched with an auxiliary memory structured as a stack.

Symbols can be inserted and extracted from the stack according to a **last-in-first-out (LIFO)** policy, i.e. the last inserted symbol is the first that is removed.

Formally, a DPA is a 6-tuple $(Q,\Sigma,\Gamma,\delta,q_0,F)$ where

1. Q is a finite set called the **states**
2. Σ is a finite set called the **alphabet**
3. Γ is a finite set called the **stack alphabet**
4. $\delta: Q \times \Sigma \times \Gamma \rightarrow Q \times \Gamma$ is the **transition function**
5. $q_0 \in Q$ is the **start state**
6. $F \subseteq Q$ is the **set of accept states**

Deterministic Pushdown Transducer (DPT)

A DPT is a DPA whose output is a string and not just *accept* or *reject*.

Formally, a DPT is a 8-tuple $(Q,\Sigma,\Gamma,\delta,q_0,F,O,\eta)$ where

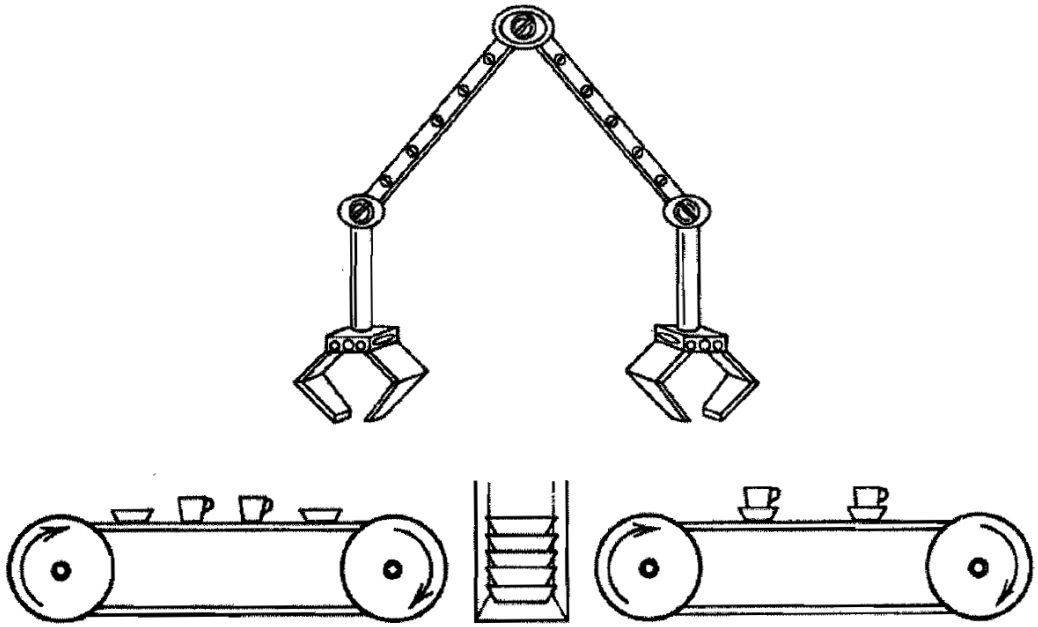
1. Q is a finite set called the **states**
2. Σ is a finite set called the **alphabet**
3. Γ is a finite set called the **stack alphabet**
4. $\delta: Q\times\Sigma\times\Gamma\rightarrow Q\times\Gamma$ is the **transition function**
5. $q_0\in Q$ is the **start state**
6. $F\subseteq Q$ is the **set of accept states**
7. O is the finite set of **output symbols**
8. $\eta: Q\times\Sigma\times\Gamma\rightarrow O$ is the **output function**

Deterministic Pushdown Transducer (DPT)

Example

A manufacturer robot that assembles saucers and small cups: if it already has a saucer and receives another saucer, it is not able to accept it and stops.

Solution: we provide the robot with a stack (initially empty).



Deterministic Pushdown Transducer (DPT)

Example

Rules:

1. If the robot has an object in one of its hands and receives another object of the same type, it puts the new object on top of the stack.

2. If the robot has an object in one of its hands and receives an object of a different type, it creates an assembled item and put it on the output conveyor belt.

3. If the robot has free hands and receives an object and the stack contains objects of the other type, the robot grasps the object on top of the stack and puts it on the output conveyor belt.

Deterministic Pushdown Transducer (DPT)

Example

The stack always contains objects of the same type, either saucers or small cups.

If the stack has infinite capacity, no FST can model this robot that has an infinite number of possible states.

Nondeterministic Pushdown Automaton (NPA)

Formally, an NPA is a 6-tuple $(Q,\Sigma,\Gamma,\delta,q_0,F)$ where

1. Q is a finite set of **states**
2. Σ is the input **alphabet**
3. Γ is the **stack alphabet**
4. $\delta: Q\times\Sigma_\varepsilon\times\Gamma_\varepsilon\rightarrow P(Q\times\Gamma_\varepsilon)$ is the **transition function**
5. $q_0\in Q$ is the **start state**
6. $F\subseteq Q$ is the **set of accept states**

where $\Sigma_\varepsilon = \Sigma\cup\{\varepsilon\}$, $\Gamma_\varepsilon = \Gamma\cup\{\varepsilon\}$ and $P(Q\times\Gamma_\varepsilon)$ is the power set of $Q\times\Gamma_\varepsilon$, i.e., the collection of all subsets of $Q\times\Gamma_\varepsilon$.

The symbol ε represents the empty string.

Context-free Languages

A context-free grammar consists of:

- a collection of substitution rules (also called **productions**)
- a set of symbols called **variables**
- a set of symbols called **terminals**

A context-free grammar generates a context-free language.

Example

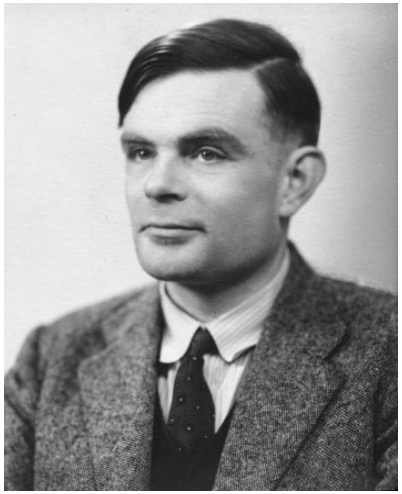
The following grammar

$$A \rightarrow 0A1$$
$$A \rightarrow B$$
$$B \rightarrow \#$$

contains three rules, two variables (A and B), and three terminals (0,1,#). This grammar generates the string 000#111, among others.

A language is **context-free** if and only if some NPA recognizes it.

Turing Machines

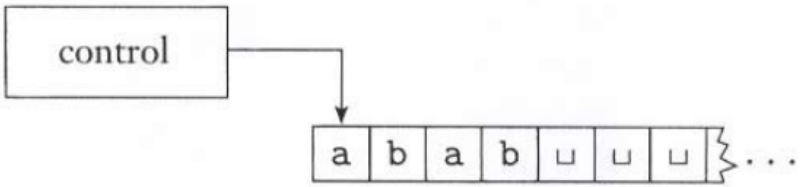


Pushdown automata are limited, because of the rigid LIFO policy that is used to manage the stack memory.

A Turing machine is a much more accurate model of a general purpose computer.

The Turing machine model uses an **infinite tape** as its unlimited memory. It has a tape head that can read and write symbols and move around on the tape.

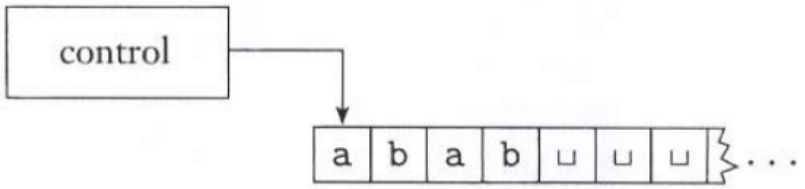
Initially the tape contains only the input string and is blank everywhere else.
To store information, the machine writes on the tape. To read information, the machine moves its head back over the cells that contain the information.



Turing Machines

At some point, the machine may stop computing and output *accept* or *reject*.

If the machine does not enter an accepting or rejecting state, it will go on forever, never halting.



Turing Machines

The collection of strings that a Turing machine M accepts is the language of M, or the language recognized by M, denoted as L(M). A language is **recognizable** if some Turing machine recognizes it.

On an input, a Turing machine may output *accept* or *reject*, but may also *loop*. A language is **decidable** if some Turing machine decides it, which means that always make a decision to accept or reject.

Turing Machines

Formally, a TM is a 7-tuple $(Q,\Sigma,\Gamma,\delta,q_0,q_{accept},q_{reject})$ where

1. Q is a finite set of **states**
2. Σ is the input **alphabet** not containing the blank symbol
3. Γ is the **tape alphabet**, containing the blank symbol and Σ
4. $\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L,R\}$ is the **transition function**
5. $q_0 \in Q$ is the **start state**
6. $q_{accept} \in Q$ is the **accept state**
7. $q_{reject} \in Q$ is the **reject state**, where $q_{reject} \neq q_{accept}$

Variants:

- multitape TM
- nondeterministic TM
- probabilistic TM

They all have an equivalent deterministic single-tape TM!

K-tape Turing Machine

Transition function: $\delta: Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L,R,S\}^k$

A *configuration* of a k-tape TM is made by the state of the control device, by the content of the k tapes and by the position of the k heads.

The *initial configuration*:

- the control device is in the initial state
- the first tape (input tape) contains a finite number of non-empty symbols
- the memory tapes contain a special symbol Z_0 in their first cell and no other non-empty symbol
- the last tape (output tape) is empty
- heads are placed on the first cell of their tapes

K-tape Turing Machine

Example

A robot that puts together a sequence of large dishes (*l*), saucers (*s*) and small cups (*c*).

Rules:

1. Receive all objects from a tape and put each object on top of 3 different stacks (for the *l*, *s*, *c*, respectively).

2. After having received all the objects, extract an *l*, an *s* and a *c* from each stack and produce an assembly (*a*).

3. At the end of the operations, check that all stacks are empty.

K-tape Turing Machine

Example

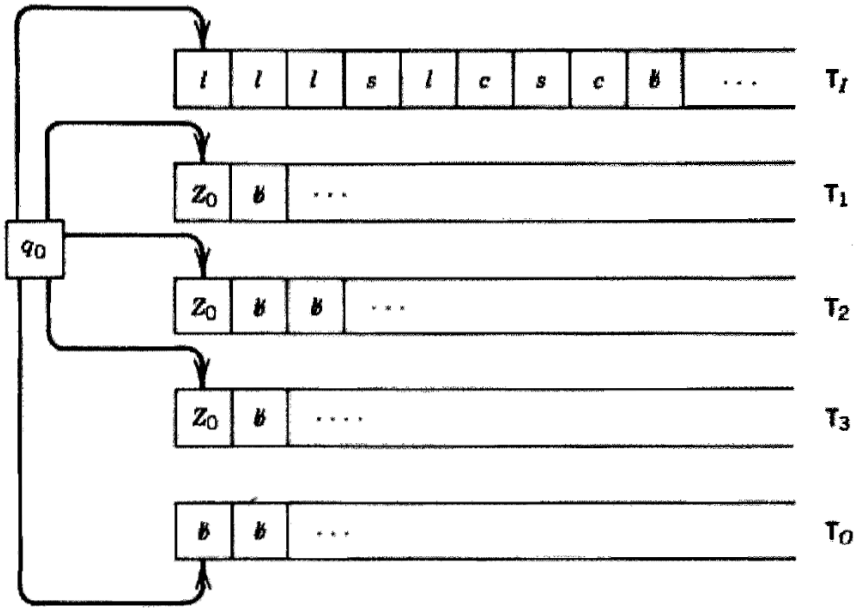
The control device of the TM that models such system has 5 states:
- q_0 (initial state)
- q_R (input tape reading state)
- q_W (output tape writing state)
- q_A (final accept state)
- q_E (final error state)

The TM has one input tape T_I, three memory tapes $\{T_1,T_2,T_3\}$ and one output tape T_O.

K-tape Turing Machine

Example

1) Start from q_0 and switch to q_R, letting unchanged the symbols Z_0 on the memory tapes and moving the corresponding heads to the right. Do not read anything from T_I and do not move the heads of T_I and T_O.



K-tape Turing Machine

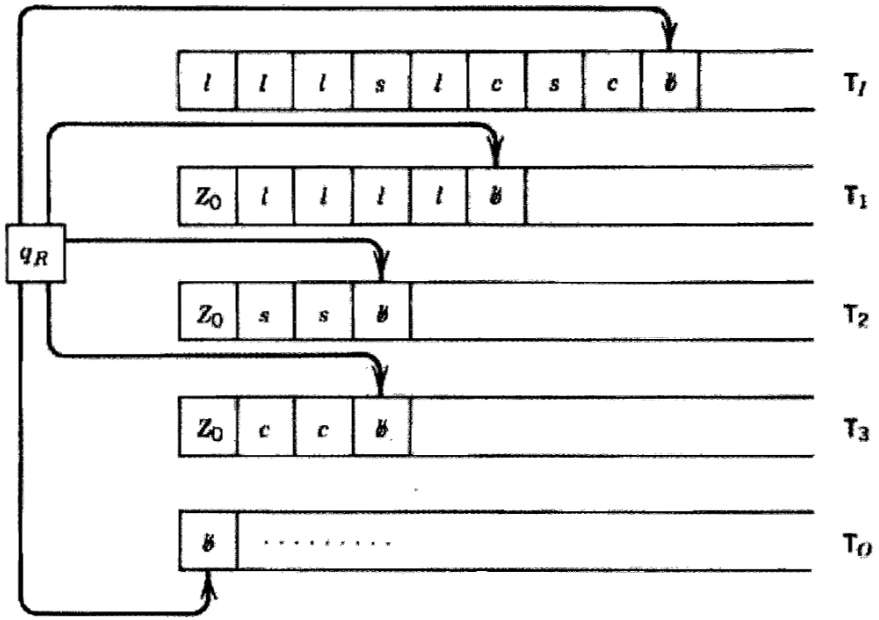
Example

2) When arriving in q_R and an *l*, *s* or *c* is read from tape T_I, stay in q_R and write input symbols on the suitable memory tape. Move the heads of T_I and of memory tapes to the right, and do not move the head of T_O.

K-tape Turing Machine

Example

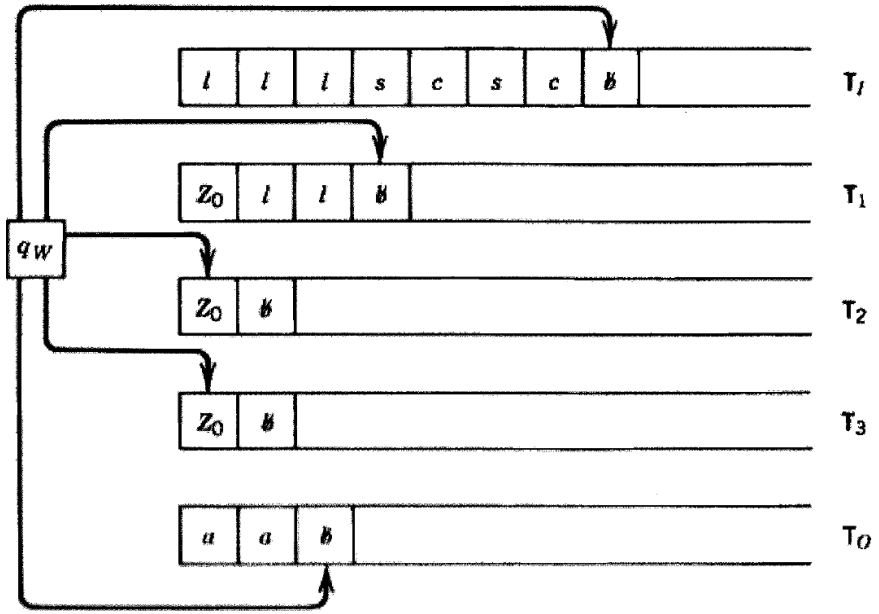
3) When arriving in q_R and an empty symbol is read from T_I, it does mean that all input objects have been collected. Do not move the head of T_I anymore, switch to q_W, move to the left the heads of the memory tapes, and do not write anything on T_O.



K-tape Turing Machine

Example

4) When arriving in q_W and symbols *l*, *s* e *c* are read from memory tapes, stay in q_W. Write an empty symbol in place of the read symbol and move the heads to the left. Moreover, write an *a* on T_O and move its head to the right.



K-tape Turing Machine

Example

5) When arriving in q_W and Z_0 is read from all memory tapes, it means that the same number of *l*, *s* and *c* has been received and that they have been assembled. Then switch to state q_A, write an empty symbol in place of Z_0 on the memory tapes, write symbol *Y* (success) on T_O, and halt.
In any other case, enter error state q_E, write symbol *N* (failure) on T_O, and halt.

Algorithms

An effective computation (**algorithm**) is a computation that produces the desired result in a finite number of steps.

Church-Turing thesis
Every effective computation can be carried out by a TM.

A function f is a **computable function** if some Turing machine M, on every input w, halts with just f(w) on its output tape.

Decidability

Certain problems can be solved algorithmically (**decided**) and others cannot.

Why should one study unsolvability?

1) Knowing when a problem is algorithmically unsolvable is useful to realize that the problem must be simplified or altered before one can find an algorithmic solution.

2) A glimpse of the unsolvable can stimulate the imagination, and help gaining an important perspective on computation.

We can formulate computational problems in terms of testing membership in a language. Showing that the language is decidable is the same as showing that the computational problem is algorithmically solvable.

Decidable Problems Concerning Regular Languages

The **acceptance problem for DFA** of testing whether a particular DFA accepts a given string can be expressed as the following language:

$A_{DFA} = \{\langle B,w\rangle \mid B \text{ is a DFA that accepts input string } w\}$

Theorem - A_{DFA} is a decidable language.

Similarly, we can define the acceptance problem for NFAs as the following language:

$A_{NFA} = \{\langle B,w\rangle \mid B \text{ is a NFA that accepts input string } w\}$

Theorem - A_{NFA} is a decidable language.

Decidable Problems Concerning Regular Languages

We can determine whether or not a DFA accepts any strings at all:

$E_{DFA} = \{\langle A \rangle \mid A \text{ is a DFA and } L(A)=\varnothing\}$

Theorem - E_{DFA} is a decidable language.

Moreover, we can determine whether two DFAs recognize the same language:

$EQ_{DFA} = \{\langle A,B \rangle \mid A \text{ and } B \text{ are DFAs and } L(A)=L(B)\}$

Theorem - EQ_{DFA} is a decidable language.

Undecidable Problems

There are several algorithmically unsolvable problems.

An example is the problem of determining whether a Turing machine accepts a given input string.

$A_{TM} = \{\langle M,w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$

Theorem - A_{TM} is undecidable.

Universal Turing Machine

Interestingly, A_{TM} is recognizable by the following TM.

U = "On input ⟨M,w⟩, where M is a TM and w is a string:

1. Simulate M on input w.
2. If M ever enters its accept state, *accept*; if M ever enters its reject state, *reject*."

If ⟨M,w⟩ is in A_{TM} then U accepts it, which actually means that U recognizes A_{TM}. Note that U loops on input ⟨M,w⟩ if M loops on w, which is why this machine does not decide A_{TM}.

U is a **Universal Turing Machine** (**UTM**), i.e., a TM that is able to simulate the evolution of any other TM from the description of that TM.

The UTM has played an important role in the development of *stored-program* computers.

Von Neumann Architecture

The von Neumann architecture is the principal model for the design of stored-program computers.

It is characterized by a central processing unit (CPU) and a memory structure that stores *both data and programs*.

Instructions are executed one after the other.

The first electronic computer based on the von Neumann architecture was EDVAC (1949).

http://web.mit.edu/STS.035/www/PDFs/edvac.pdf

Von Neumann Architecture

The scheme is based on five basic components:

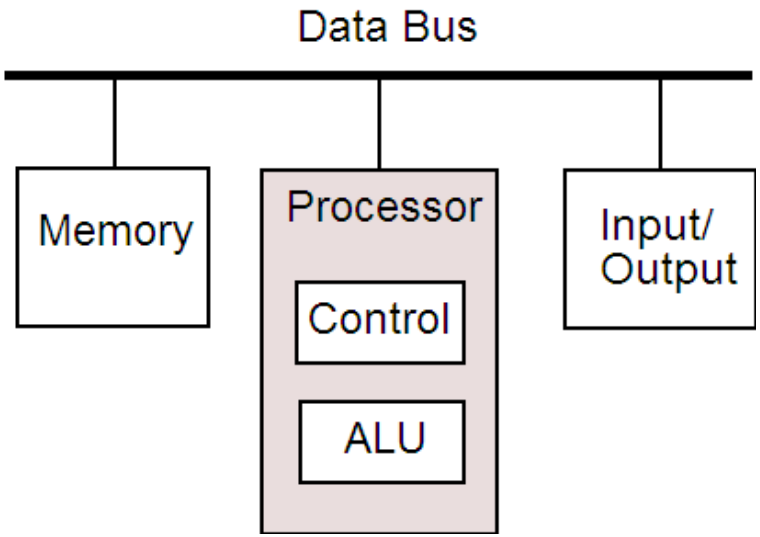
1. **CPU** or processor, which includes
   1. Arithmetic Logic Unit (ALU)
   2. Control Unit

2. **Memory**, that contains data to be processed and instructions to be executed (programs)

3. **Input unit**, through which data are inserted in the computing machine to be processed

4. **Output unit**, which is necessary to return processed data to the user

5. **Data Bus**, a channel that connect all components



Halting Problem

The undecidability of A_{TM} can be used to prove the undecidability of the halting problem.

$HALT_{TM} = \{\langle M,w\rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$

Turing's halting theorem - $HALT_{TM}$ is undecidable.

PROOF. Let us assume that R is a TM that decides $HALT_{TM}$. We construct S, a TM to decide A_{TM}:

S = "On input ⟨M,w⟩, where M is a TM and w is a string:

1. Run R on input ⟨M,w⟩.
2. If R rejects, *reject*.
3. If R accepts, simulate M on w until it halts.
4. If M has accepted, *accept*; if M has rejected, *reject*."

Clearly, if R decides $HALT_{TM}$, then S decides A_{TM}. This is impossible! □

Halting Problem

An equivalent way to state the halting theorem is the following one.

Turing's halting theorem - There is no program P able to predict whether any other program Q (given as input to P) will stop after a finite number of steps or not.

PROOF. Suppose by contradiction that such a program *P* exists. Then, we can modify *P* to produce a new program *P'* that, given another program *Q* as input, does the following:

1. runs forever if *Q* halts
2. halts if *Q* runs forever

In this way, if we feed *P'* its own code as input, then *P'* will run forever if it halts, or halt if it runs forever.

Therefore, *P'* (and *P*, by implication) cannot exist. □

Halting Problem

The halting theorem tells us that the general problem of software verification (i.e., verifying that a program performs as specified) is algorithmically unsolvable.

A program specification *usually* includes a statement to the effect that the program is supposed to produce an answer of some sort.

But checking whether the program eventually produces a result is equivalent to the halting problem, which is undecidable.

Reducibility

The primary method for proving that problems are computationally unsolvable is called **reducibility**.

A **reduction** is a way of converting one problem to another problem in such a way that a solution to the second problem can be used to solve the first problem.

Example:
- you want to find your way around a new city
- you know that doing so would be easy if you had a map
- thus, you can reduce the first problem (finding your way around the city) to the second problem (finding a map)

Reducibility and Decidability

Reducibility always involves two problems, which we call A and B.

If A is reducible to B and B is decidable, A also is decidable.
Equivalently, if A is undecidable and reducible to B, B is undecidable.

We often show that a problem is undecidable by showing that the halting problem reduces to that problem.

If A is reducible to B and B is reducible to A, we say that A and B are equivalent problems.

Mapping Reducibility

There are several ways of defining the notion of reducing one problem to another. A simple type of reducibility is called mapping reducibility.

As usual, we represent computational problems by languages.

Language A is **mapping reducible** to language B, written $A \leq_m B$, if there is a computable function f, where for every w,

$$w \in A \Leftrightarrow f(w) \in B.$$

The function f is called the **reduction** of A to B.

Turing Reducibility

Turing reducibility is a generalization of mapping reducibility that captures our intuitive concept of reducibility more closely.

An **oracle** for language B is an external device that is capable to decide language B, i.e., to report whether any string w is a member of B.

An **oracle Turing machine** is a modified TM that has the additional capability of querying an oracle and getting the answer in a single computation step. We write M^B to describe an oracle Turing machine that has an oracle for language B.

Language A is **Turing reducible** to language B, written $A \leq_T B$, if A is **decidable relative to** B, which means that there is an oracle Turing machine M^B that is able to decide A.

Turing Reducibility

Theorem - If $A \leq_T B$ and B is decidable, then A is decidable.

PROOF. If B is decidable, then we may replace the oracle for B by an actual procedure that decides B. Thus, we may replace the oracle Turing machine that decides A by an ordinary TM that decides A. □

Turing reducibility is a generalization of mapping reducibility. Indeed, the mapping reduction may be used to give an oracle Turing machine that decides A relative to B.

$A \leq_m B \Rightarrow A \leq_T B$ (not vice versa!)

Turing Reducibility

Example

Consider the languages A_{TM} and $\overline{A_{TM}}$.

Intuitively, they are reducible one to another (a solution to either would be used to solve the other by reversing the answer).

Indeed, they are Turing reducible one to another.

However, $\overline{A_{TM}}$ is not mapping reducible to A_{TM} because A_{TM} is Turing-recognizable but $\overline{A_{TM}}$ is not (if it was, then A_{TM} would be decidable, which is impossible).

References

• M. Sipser, *Introduction to the Theory of Computation*, 3rd ed., Cengage Lerning, 2013

• SIGACT https://www.sigact.org/

• EATCS http://eatcs.org/

• Turing Award https://amturing.acm.org/